Verified Optimizations for the Intel* IA-64 Architecture**

Jim Grundy

The Australian National University
Department of Computer Science, Canberra ACT 0200, Australia
Jim.Grundy@acm.org

Abstract. This paper outlines a formal model of the Intel IA-64 architecture, and explains how this model can be used to verify the correctness of assembly-level code optimizations. The formalization and proofs were carried out using the HOL Light theorem prover.

1 Introduction

Current microprocessors dynamically reorder the sequence of instructions being executed to extract greater performance. The IA-64 takes a different approach [3, 6]. By exposing architectural features that would ordinarily be hidden, IA-64 allows the compiler to reorder instructions prior to execution. By moving the burden of instruction reordering from hardware to software, resources are freed to increase performance in other ways.

Formal methods have been applied to instruction reordering hardware [7], but as the responsibility for reordering instructions moves from hardware to software, so does the obligation to ensure the reorderings preserve the meaning of the code. This paper proves the correctness of some of the instruction reorderings performed in software for the IA-64. The work described deliberately stops-short of tackling the open-ended difficulty of verifying general properties of IA-64 programs. Instead, the proofs are limited to checking equivalence between similar small programs; the kinds of proofs that typify verification of individual optimizing transformations. The purpose of the work is to investigate the extent to which such proofs can be automated. The formalization and proofs were carried out with the HOL Light theorem prover [5].

2 Examples: Control and Data Speculation

This paper will focus on two examples of instruction reordering. The first illustrates *control speculation*, where an instruction is executed even though the result is not known to be needed. The second example illustrates *data speculation*, where an instruction is executed using data not known to be accurate.

---

* All names and brands are the property of their respective owners.

** Research supported by the Intel Corporation and the Australian Research Council.

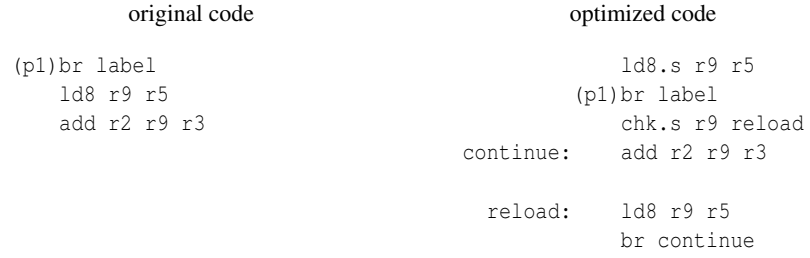

Fig. 1. Control speculation example

2.1 Control Speculation

The execution of most IA-64 instructions may be predicated on the value of a one-bit *predicate* register. If the nominated register holds true, the instruction executes normally; if not, it has no effect. Predicated instructions are written with the predicate register parenthesized to the left. An example can be seen in the first instruction of the 'original' code fragment in Fig. 1.

Consider the original code presented in Fig. 1. If `p1` holds true, then control branches to `label`. If not, execution falls through to the next instruction, which loads the general purpose register `r9` with 8 bytes from the address held in `r5`. The values held in `r9` and `r3` are summed, and the result stored in `r2`.

Load instructions take several cycles to complete, and in this program the load is followed by an add which depends upon the value loaded. The execution of the add must therefore stall, to allow the load to complete before it can execute. We would like hide the load latency by moving the load earlier in the instruction stream. Unfortunately, we cannot execute the load earlier as it appears immediately after a conditional branch; if the branch is taken the load should not be executed. It is tempting to think we could move the load before the branch and ignore the result if it is not needed; this would be a *control speculative* execution of the load. However, this could cause a fault if the load tried to access an invalid address. A correct, but unnecessary, load could also incur a performance penalty if it required nonresident memory to be swapped in.

On a traditional architecture the load would stay where it was, but the IA-64 offers a way around these problems. Every register has a corresponding one-bit tag called a *not a thing* (nat) bit. A control speculative version of the load instruction is provided, which quietly sets this bit rather than causing a fault (including a page fault). The nat bit can be checked to see if the load succeeded. Using this feature, the code can be optimized as shown in Fig. 1.

The optimized code begins with a control speculative load, which attempts to read data into `r9`. If the load fails, the nat bit of `r9` is set. Later, the `chk.s` instruction checks if `r9` contains valid data. If the nat bit is clear, the load succeeded and execution continues unaffected. If it is set, execution is transferred to the code labeled `reload`, where the load is retried. The second load will exhibit the true faulting behavior, perhaps causing nonresident memory to be swapped in so the load can complete. Both paths then execute the add instruction.

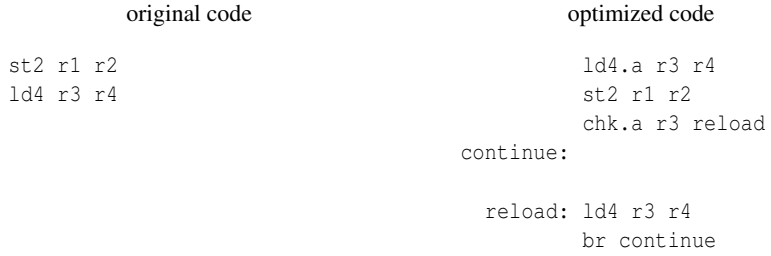

Fig. 2. Data speculation example

2.2 Data Speculation

Figure 2 gives another example optimization. Consider the unoptimized code. The first instruction stores two bytes from `r2` to the address held in `r1`. The second loads four bytes into `r3` from the address held in `r4`. As before, we would like hide the load latency from any subsequent instructions by moving the the load earlier in the instruction stream. However, this would result in incorrect behavior if the memory region written by the store overlaps that read by the load. In most situations the regions will not overlap, but they might, so the load must remain after the store.

The IA-64 provides a way around this obstacle with a special data speculative *advanced* load instruction. The advanced load records the region of memory a register was loaded from. A test can be used to check if the region has been overwritten since the load. Using this feature, the code can be optimized as shown. The optimized code begins with an advanced load, followed by the store. Next, register `r3` is checked to see if it was effected by the store. If it was, a branch is taken to the label `reload`, where it is reloaded with the correct data.[1]

3 A Model of the IA-64

Our aim is to describe an abstract model of an IA-64 machine that can be used to show that the optimized code fragments just presented have the same behavior as the corresponding unoptimized fragments. This section will describe each component of the IA-64 architectural state that needs to be modeled to verify these optimizations.

3.1 Data Memory

We make the simplifying assumption that the instruction and data memories can be modeled separately. The data memory is defined in terms of two types, *word* and *size*. The type *word* describes 64-bit words, which are used to hold both addresses and data. The type *size* describes the units in which memory is accessed: 1, 2, 4 or 8 bytes.

$$\mathit{word} \stackrel{\text{def}}{=} \{n \mid n < 2^{64}\} \qquad \mathit{size} \stackrel{\text{def}}{=} \{1,2,4,8\}$$

[1] Other IA-64 instructions can handle these simple examples more succinctly. Here we present only the most general forms of speculation and recovery.

The function zext takes a word w and a size s, and returns a new word with a zero-extended copy of the first s bytes of w.

$$\vdash^{\mathsf{def}} \mathsf{zext}\ w\ s = w \bmod 2^{8\times s} \qquad \text{[zext\_def]}$$

A word (address) and a size together describe a *region* of memory. The predicate overlapped determines if two regions overlap.

$$\vdash^{\mathsf{def}} \mathsf{overlapped}\ a_1\ s_1\ a_2\ s_2 = \qquad \text{[overlapped\_def]}$$
$$(\exists x \cdot a_1 \le x \wedge x < a_1 + s_1 \wedge a_2 \le x \wedge x < a_2 + s_2)$$

The data memory is modeled as a function from words (addresses) to words (values). The mem_read and mem_write operations are described as follows:

$$\vdash^{\mathsf{def}} \mathsf{mem\_read}\ m\ a\ s = \mathsf{zext}\ (m\ a)\ s \qquad \text{[mem\_read\_def]}$$

$$\vdash^{\mathsf{def}} (\mathsf{mem\_read}\ (\mathsf{mem\_write}\ m\ a\ s\ w)\ a\ s = \mathsf{zext}\ w\ s) \wedge \qquad \text{[mem\_write\_def]}$$
$$(\neg \mathsf{overlapped}\ a_1\ s_1\ a_2\ s_2 \Longrightarrow$$
$$\mathsf{mem\_read}\ (\mathsf{mem\_write}\ m\ a_2\ s_2\ w)\ a_1\ s_1 = \mathsf{mem\_read}\ m\ a_1\ s_1)$$

Note that the behavior of reads and writes that access overlapping, but not identical, regions of memory is unspecified. Accurate modeling of such accesses is not necessary to verify optimizations like the ones discussed here.

Not all regions of memory are valid sources or destinations, this includes those that extend outside the address space, but may include others as well.

$$\vdash^{\mathsf{def}} \mathsf{mem\_valid\_source}\ a\ s \Longrightarrow a + s \le 2^{64} \qquad \text{[mem\_valid\_source\_def]}$$

$$\vdash^{\mathsf{def}} \mathsf{mem\_valid\_dest}\ a\ s \Longrightarrow a + s \le 2^{64} \qquad \text{[mem\_valid\_dest\_def]}$$

Some, *sequential*, regions of memory should be accessed only in the order originally specified. If, for example, IO devices are mapped into the memory space, those regions will be sequential. We do not specify which regions of memory are sequential, only that some may be.

$$\vdash^{\mathsf{def}} \mathsf{mem\_seq}\ a\ s \Longrightarrow \mathsf{T} \qquad \text{[mem\_seq\_def]}$$

Not all regions of memory may be read speculatively. This includes invalid sources and sequential regions, but may include other regions as well. An obvious example is memory that is nonresident and would therefore need to be swapped in. The validity of speculatively accessing a memory region may change as the state of the machine changes. The mem_valid_spec_source predicate takes an extra parameter x to represent the abstract state of the machine. It is not necessary to specify how the value of mem_valid_spec_source depends on x, only that it may, and that the type of x is sufficiently large to encompass the state of the machine.

$$\vdash^{\mathsf{def}} \mathsf{mem\_valid\_spec\_source}\ x\ a\ s \Longrightarrow \qquad \text{[mem\_valid\_spec\_source\_def]}$$
$$\mathsf{mem\_valid\_source}\ a\ s \wedge \neg \mathsf{mem\_seq}\ a\ s$$

Memory Access Ordering: It is not always possible to reorder memory accesses as described in Sect. 2.2. Code to synchronize multiple processes may depend on the precise ordering of those accesses. Changes to the memory access ordering that appear correct when viewed from the perspective of an individual process, may not be correct when the collection of processes are considered as a whole. The IA-64 provides special variants of the load and store instructions, and a special 'memory fence' instruction for use in such routines. These instructions must respect the memory access ordering. The execution of the ordinary load and store instructions are not required to access memory in the order they were issued. The memory accesses may be reordered or even coalesced by the hardware, provided that the resulting access order satisfies read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) data dependencies [6]. The optimizations considered here use only the ordinary versions of the load and store instructions, and so the memory model presented does not address access ordering. Mike Gordon has described a more elaborate memory model that encompasses memory access ordering issues for the Alpha architecture [4].

3.2 General Purpose Registers

The IA-64 architecture defines 128 general purpose registers. Each register holds a 64-bit word and a one-bit tag called a *not a thing* (nat) bit. The role of the nat bit is to indicate when the data held in the register is invalid due to a failed control speculation. These bits are set by failing control speculative loads, and are propagated by operations that use invalid data as input. We will describe the contents of a register with a record type:[2]

$$\mathit{register} \stackrel{\text{def}}{=} \lhd \mathsf{val}: \mathit{word}; \mathsf{nat}: \mathit{bool} \rhd$$

These registers cannot necessarily all be accessed by the instructions of a particular routine. Each routine has access to a subset of the registers known as a *frame*. The current frame moves though the register file as subroutines are entered and exited. This is similar to the register window system of the SPARC architecture [9], except that IA-64 frames may be of variable size. Hardware automatically renames the registers so that the current frame appears at the start of the register file. Even though the optimizations described here do not involve subroutine calls, the description of the register frame mechanism cannot be completely ignored. Within a routine, attempts to write registers not in the current frame will cause a fault, while reading such registers will produce undefined results.

A new type *grindex* is defined for the set of general purpose register indexes, and we define a constant sof (size of frame) of that type to model the size of the current frame. The actual value of sof is unimportant, except that all frames must contain at least 32 registers.[3]

---

[2] The actual formalization uses tuples as records are not supported in HOL Light. Records, in the style of hol98 [8], have been used here to simplify the presentation.

[3] The size of frame (sof) is defined as a constant because its value is not changed by the instructions used in the examples presented here. In general, however, its value can change and is better modeled as part of the state space defined in Sect. 3.5.

$$\mathit{grindex} \stackrel{\mathsf{def}}{=} \{n \mid n < 128\} \qquad \vdash^{\mathsf{def}} 32 \leq \mathsf{sof} \qquad [\mathsf{sof\_def}]$$

We define predicates reg_valid_source and reg_valid_dest to indicate which registers may be read and written.

$$\vdash^{\mathsf{def}} \mathsf{reg\_valid\_source}\ i = i < \mathsf{sof} \qquad [\mathsf{reg\_valid\_source\_def}]$$

$$\vdash^{\mathsf{def}} \mathsf{reg\_valid\_dest}\ i = \mathsf{reg\_valid\_source}\ i \wedge (i \neq 0) \qquad [\mathsf{reg\_valid\_dest\_def}]$$

The definition of the read and write operations for registers is straight forward; the only complications being due to the under-determined value of invalid reads and the hard-wired value of register 0. Note that the parameter x, as before, is used to allow the result of an undefined read to depend on some abstract notion of the general machine state.

$$\vdash^{\mathsf{def}} \mathsf{reg\_valid\_source}\ i \Longrightarrow \qquad [\mathsf{reg\_read\_def}]$$
$$\mathsf{reg\_read}\ x\ f\ i = \mathsf{if}\ i = 0\ \mathsf{then} \lhd \mathsf{val}:= 0; \mathsf{nat}:= \mathsf{F} \rhd\ \mathsf{else} f\ i$$

$$\vdash^{\mathsf{def}} \mathsf{reg\_write}\ f\ i\ v = (\lambda j \cdot \mathsf{if}\ j = i\ \mathsf{then}\ v\ \mathsf{else}\ f\ j) \qquad [\mathsf{reg\_write\_def}]$$

It was not necessary to under-specify the result of invalid writes, because IA-64 instructions raise a fault rather than attempt this operation.

The following basic theorems regarding register operations are necessary stepping stones to verifying the optimizations.

$$\vdash \mathsf{reg\_read}\ x\ f\ 0 = \lhd \mathsf{val}:= 0; \mathsf{nat}:= \mathsf{F} \rhd \qquad [\mathsf{reg\_read\_zero\_thm}]$$

$$\vdash \mathsf{reg\_valid\_dest}\ i \Longrightarrow \mathsf{reg\_read}\ x\ (\mathsf{reg\_write}\ f\ i\ v)\ i = v \qquad [\mathsf{reg\_read\_eq\_thm}]$$

$$\vdash \mathsf{reg\_valid\_source}\ i \wedge i \neq j \Longrightarrow \qquad [\mathsf{reg\_read\_ne\_thm}]$$
$$\mathsf{reg\_read}\ x\ (\mathsf{reg\_write}\ f\ j\ v)\ i = \mathsf{reg\_read}\ x\ f\ i$$

$$\vdash \mathsf{reg\_write}\ (\mathsf{reg\_write}\ f\ i\ v)\ i\ w = \mathsf{reg\_write}\ f\ i\ w \qquad [\mathsf{reg\_write\_eq\_thm}]$$

3.3 Predicate Registers

The IA-64 includes 64 one-bit predicate registers. These registers can be used to mask execution of individual instructions. If the execution of an instruction is conditional on the value of a predicate register, then the instruction is said to be *predicated* on that register. The description of the predicate registers is simpler than that of the general purpose registers as there is no notion of frame, the predicate registers are always visible, and their contents are always valid. Strictly speaking, (almost) all IA-64 instructions are predicated, those which are to be executed unconditionally are predicated on register 0, which is hard-wired to true. Writes to predicate register 0 are allowed, but they have

no visible effect on the state. The operators on the predicate registers are pred_read and pred_write, and their definitions are similar to those for reg_read and reg_write.

$$\mathit{prindex} \stackrel{\mathsf{def}}{=} \{n \mid n < 64\}$$

$$\vdash^{\mathsf{def}} \mathsf{pred\_read}\ f\ i = \mathsf{if}\ i = 0\ \mathsf{then}\ \mathsf{T}\ \mathsf{else}\ f\ i \qquad \text{[pred\_read\_def]}$$

$$\vdash^{\mathsf{def}} \mathsf{pred\_write}\ f\ i\ b = (\lambda j \cdot \mathsf{if}\ i \neq 0 \wedge j = i\ \mathsf{then}\ b\ \mathsf{else}\ f\ j) \qquad \text{[pred\_write\_def]}$$

Properties similar to those proved about reg_read and reg_write hold for pred_read and pred_write as well.

$$\vdash \mathsf{pred\_read}\ f\ 0 = \mathsf{T} \qquad \text{[pred\_read\_0\_thm]}$$

$$\vdash i \neq 0 \Longrightarrow \mathsf{pred\_read}\ (\mathsf{pred\_write}\ f\ i\ b)\ i = b \qquad \text{[pred\_read\_eq\_thm]}$$

$$\vdash i \neq j \Longrightarrow \mathsf{pred\_read}\ (\mathsf{pred\_write}\ f\ j\ b)\ i = \mathsf{pred\_read}\ f\ i \qquad \text{[pred\_read\_ne\_thm]}$$

$$\vdash \mathsf{pred\_write}\ (\mathsf{pred\_write}\ f\ i\ b)\ i\ c = \mathsf{pred\_write}\ f\ i\ c \qquad \text{[pred\_write\_eq\_thm]}$$

3.4 The ALAT

Data speculative, or advanced, load instructions must keep track of the integrity of data that has been loaded. Any subsequent stores overlapping the region loaded will invalidate the data. On the IA-64 this task is performed using an architectural feature called the *Advanced Load Address Table* (ALAT).

Ideally, the ALAT records the following information for each speculatively loaded register:

- Whether the data in the register is valid.
- If so, what region of memory the data was loaded from.

The ALAT entry for each register can be described as a record as follows:

$$\mathit{alat\_entry} \stackrel{\mathsf{def}}{=} \langle\!\triangleleft \mathsf{valid}: \mathit{bool}; \mathsf{addr}: \mathit{word}; \mathsf{sz}: \mathit{size} \triangleright\!\rangle$$

The information recorded in the ALAT does not have to be completely accurate to ensure the correct behavior of IA-64 programs. If the ALAT records a register as holding valid data, then that data must be valid; but the ALAT may falsely record that the data in a register is invalid. Such inaccuracy could cause suboptimal performance as it may force valid data to be reloaded, but the functional behavior of the program should be unaltered. There are many reasons why a particular implementation of the ALAT might exhibit such inaccuracy. For example, the ALAT may have fewer entries than there are registers. In order to capture the full generality of potential ALAT implementations, the specification presented gives only those properties that must be honored to

guarantee correct program execution. This specification allows the ALAT to lose information in a variety of controlled ways. Indeed, an empty table would trivially satisfy the specification, though it would make for inefficient execution.

An ALAT will be modeled as a function from register indices to ALAT entries. The simplest operation on the ALAT does nothing except allow the ALAT to forget about the validity of one or more registers. This operation is called leak.

$$\vdash^{\mathsf{def}} (\neg(t\ i).\mathsf{valid} \Longrightarrow \neg(\mathsf{leak}\ t\ i).\mathsf{valid}) \wedge \quad [\mathsf{leak\_def}]$$
$$((\mathsf{leak}\ t\ i).\mathsf{valid} \Longrightarrow \mathsf{leak}\ t\ i = t\ i)$$

The first clause of the definition asserts that the leak operation will not cause an invalid register to become valid. The second clause states that any register still valid after the leak has the same ALAT entry it had before.

A more constructive operation attempts to add information to the ALAT. The function validate $t\ i\ a\ s$ attempts to add to the ALAT t the fact that register i contains valid data loaded from the region with address a and size s. It might not succeed, and it may cause the ALAT to forget about other registers.

$$\vdash^{\mathsf{def}} (\neg(t\ j).\mathsf{valid} \Longrightarrow \neg(\mathsf{validate}\ t\ i\ a\ s\ j).\mathsf{valid} \vee i = j) \wedge \quad [\mathsf{validate\_def}]$$
$$((\mathsf{validate}\ t\ i\ a\ s\ i).\mathsf{valid} \Longrightarrow$$
$$\mathsf{validate}\ t\ i\ a\ s\ i = \lhd \mathsf{valid} := \mathsf{T}; \mathsf{addr} := a; \mathsf{sz} := s \rhd) \wedge$$
$$(i \neq j \wedge (\mathsf{validate}\ t\ i\ a\ s\ j).\mathsf{valid} \Longrightarrow \mathsf{validate}\ t\ i\ a\ s\ j = t\ j)$$

The first clause of this definition asserts that validating a register i will not cause any other register to become valid. The second clause asserts that if after validating register i, it is indeed valid, then i has associated with it the address and size supplied. The final clause states that if any other register is valid after the operation, the entry associated with it is unchanged.

The expression invalidate_single $t\ i$ represents invalidating an individual register i from an ALAT t. Using invalidate_single may also invalidate other registers.

$$\vdash^{\mathsf{def}} (\neg(t\ j).\mathsf{valid} \Longrightarrow \neg(\mathsf{invalidate\_single}\ t\ i\ j).\mathsf{valid}) \wedge \quad [\mathsf{invalidate\_single\_def}]$$
$$((\mathsf{invalidate\_single}\ t\ i\ j).\mathsf{valid} \Longrightarrow i \neq j \wedge \mathsf{invalidate\_single}\ t\ i\ j = t\ j)$$

The operation invalidate_multiple $t\ a\ s$ has the effect of invalidating in ALAT t all those registers loaded from regions that overlap the region with address a and size s. Other entries may also be invalidated as a result of this operation.

$$\vdash^{\mathsf{def}} (\neg(t\ i).\mathsf{valid} \Longrightarrow \quad [\mathsf{invalidate\_multiple\_def}]$$
$$\neg(\mathsf{invalidate\_multiple}\ t\ a\ s\ i).\mathsf{valid}) \wedge$$
$$((\mathsf{invalidate\_multiple}\ t\ a\ s\ i).\mathsf{valid} \Longrightarrow$$
$$\neg\mathsf{overlapped}\ (t\ i).\mathsf{addr}\ (t\ i).\mathsf{sz}\ a\ s \wedge \mathsf{invalidate\_multiple}\ t\ a\ s\ i = t\ i)$$

Further ALAT Freedoms: An ALAT has the freedom to lie, in a conservative way, about the information it records. An ALAT may report that a register contains invalid data, even when it records that the data is valid. A subsequent query about the register may correctly answer that the data is valid. To model this behavior, we need another function to check the validity of a register.

$$\vdash^{\mathsf{def}} \mathsf{check}\ x\ t\ i \Longrightarrow (t\ i).\mathsf{valid} \qquad \text{[check\_def]}$$

The important features of check are as follows:

- If check reports that a register is valid, then it really is valid.
- The value returned by check depends, in an unspecified way, on a variable x that represents an abstraction of the entire machine state.

3.5 The Machine State

The whole machine is modeled as a record of the components described thus far:

$$\begin{array}{lll}
\mathit{state} \stackrel{\mathsf{def}}{=} \lhd \mathsf{ip}: \mathit{num}; & - \text{ instruction pointer} \\
\quad \mathsf{mem}: \mathit{word} \to \mathit{word}; & - \text{ data memory} \\
\quad \mathsf{grfile}: \mathit{grindex} \to \mathit{register}; & - \text{ general purpose register file} \\
\quad \mathsf{prfile}: \mathit{prindex} \to \mathit{bool}; & - \text{ predicate register file} \\
\quad \mathsf{alat}: \mathit{grindex} \to \mathit{alat_entry}; & - \text{ advanced load address table} \\
\quad \mathsf{unknown}: \mathit{ind} \rhd & - \text{ other unknown state}
\end{array}$$

Two components of the state record were not previously alluded to. The instruction pointer ip stores the location of the current instruction in a separate instruction memory. The unknown field represents an abstraction of the other aspects of the state of an IA-64 that are not modeled here. Its purpose is to serve as an argument to under-specified functions where the result may depend on things other than those components of the state that have been modeled concretely. The type *ind* is used for unknown because little is known about it, except that it is large enough to encode a representation of the complete machine state.

4 IA-64 Instruction Semantics

To verify the optimizations presented in Sect. 2 we need to model the effect of executing an IA-64 program until a particular point in the code is reached. We will model the outcome of doing this with a new type *outcome*.

$$\begin{array}{lll}
\mathit{outcome} \stackrel{\mathsf{def}}{=} \mathsf{STATE}\ \mathit{state} & - \text{ reaches nominated state} \\
\quad \mid \mathsf{FAULT}\ \mathit{fault} & - \text{ faults before reaching nominated state} \\
\quad \mid \bot & - \text{ neither faults nor reaches nominated state}
\end{array}$$

The type *fault* describes IA-64 faults visible to applications programmers (i.e., page faults are not included).

$$\mathit{fault} \stackrel{\mathsf{def}}{=} \mathsf{NAT\_CONSUMPTION} \mid \mathsf{ILLEGAL\_OPERATION} \ldots$$

The meaning of each IA-64 instruction can be specified as a function from an initial state to an outcome. An example giving the definition of the advanced load instruction can be found in Fig. 3. The form of the definition is similar to that of the C pseudo code that defines this instruction in the architecture guide [6]. A type encompassing all IA-64 instructions can now be defined.

```
⊢def ld_a p s r1 r2 σ =
    let addr = reg_read σ.unknown σ.grfile r2 in
    let data = mem_read σ.mem addr.val s in
      if ¬pred_read σ.prfile p then
        STATE σ
      else if ¬reg_valid_dest r1 then
        FAULT ILLEGAL_OPERATION
      else if addr.nat then
        FAULT NAT_CONSUMPTION
      else if ¬mem_valid_source addr.val s then
        FAULT ILLEGAL_LOAD
      else if mem_seq addr.val s then
        STATE σ with
                ◁grfile:= reg_write σ.grfile r1 ◁val:= 0; nat:= F▷;
                  alat:= invalidate_single σ.alat r1▷
      else
        STATE σ with
                ◁grfile:= reg_write σ.grfile r1 ◁val:= data; nat:= F▷;
                  alat:= validate σ.alat r1 addr.val s▷
```

p:	predicate
s:	size of data to load
r_1:	destination register
r_2:	register with source address
σ:	initial state

Fig. 3. Meaning of the advanced load instruction

```
inst ≝ LD prindex size grindex grindex
     | LD_A prindex size grindex grindex
     | LD_S prindex size grindex grindex
     | CHK_A prindex grindex num
     ⋮
```

Assuming that we have a complete set of instruction meanings in the style of Fig. 3, we can define a function mapping instructions to their meaning.

```
⊢def ⟦LD p s r1 r2⟧ = ld p s r1 r2 ∧                    [inst_sem_def]
   ⟦LD_A p s r1 r2⟧ = ld_a p s r1 r2 ∧
   ...
```

Some actions are common to all instructions and are therefore factored out. In particular, each instruction should advance the instruction pointer and change the unknown component of the state in some unspecified way. We define a function to return the unknown component of the next state, based on the current state σ and the instruction i. This function is completely unspecified.

```
⊢def next_unknown σ i = x ⟹ T                          [next_unknown_def]
```

We can now define a function step to advance the execution of a program in an instruction memory p by one step. Should an instruction cause a fault, step will make no further progress.

$$\vdash^{\mathsf{def}} \mathsf{step}\ p\ (\mathsf{STATE}\ \sigma) = \qquad \text{[step\_def]}$$
$$[\![p\ \sigma.\mathsf{ip}]\!](\sigma\ \mathsf{with}\ \lhd \mathsf{ip}{:=}\ \sigma.\mathsf{ip}+1;$$
$$\mathsf{unknown}{:=}\ \mathsf{next\_unknown}\ \sigma\ (p\ \sigma.\mathsf{ip})\rhd) \wedge$$
$$\mathsf{step}\ p\ (\mathsf{FAULT}\ f) = \mathsf{FAULT}\ f$$

4.1 Execution Sequences

The examples presented in Sect. 2 compare two programs by posing the question: Is the effect of one program when executed until it reaches some nominated instruction the same as that of another program when it is executed until it reaches a nominated instruction? To answer this question, we need to formalize what it means to execute a program until a nominated instruction is reached.

The first thing to note is that some executions of a program will raise faults, and therefore never reach a particular target instruction. To be more precise then, we are interested in what it means to execute a program until some nominated instruction is reached or a fault is raised. If l is the location of the instruction we are interested in, then the predicate $\mathsf{at\_or\_fault}\ l$ describes those outcomes where we have reached our goal.

$$\vdash^{\mathsf{def}} \mathsf{at\_or\_fault}\ l\ (\mathsf{STATE}\ \sigma) = (\sigma.\mathsf{ip} = l) \wedge \qquad \text{[at\_or\_fault\_def]}$$
$$\mathsf{at\_or\_fault}\ l\ (\mathsf{FAULT}\ f) = \mathsf{T}$$

A program may contain loops, so during its execution it may execute the same instruction many times. When we talk about executing a particular program until a given instruction is reached, we are interested in the *first* time that instruction is reached. It simplifies the formalization to introduce a binder function that captures the concept of being 'the first.' We introduce the notation '$\boldsymbol{\varepsilon}_1 n{\cdot}P\ n$' to represent the first number for which P holds. For example, $(\boldsymbol{\varepsilon}_1 n{\cdot}n > 10)$ is 11.

$$\vdash^{\mathsf{def}} (\exists n{\cdot}P\ n) \Longrightarrow \qquad [\boldsymbol{\varepsilon}_1\text{\_def}]$$
$$P\ (\boldsymbol{\varepsilon}_1 n{\cdot}P\ n) \wedge (\forall m{\cdot}m < (\boldsymbol{\varepsilon}_1 n{\cdot}P\ n) \Longrightarrow \neg P\ m)$$

Using $\boldsymbol{\varepsilon}_1$, we can define the expression $(f\ \mathsf{until}\ P)\ o$ to represent repeatedly applying the function f to o until some desired outcome, characterized by P, is reached. This expression yields $\perp$ if the desired outcome can never be reached.

$$\vdash^{\mathsf{def}} (f\ \mathsf{until}\ P)\ o = \mathsf{if}\ (\exists n{\cdot}P\ (f^n\ o))\ \mathsf{then}\ f^{(\boldsymbol{\varepsilon}_1 n{\cdot}P\ (f^n\ o))}\ o\ \mathsf{else}\ \perp \qquad \text{[until\_def]}$$

We can now phrase as follows the meaning of 'executing the program in p until the instruction at l is reached.'

$$(\mathsf{step}\ p)\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ l)$$

4.2 Reasoning about until

The following theorem allows us to reason about IA-64 programs using a form of symbolic simulation within the theorem prover. It allows us to take repeated steps in the program until we reach the desired outcome.

$$\vdash ((\mathsf{step}\ p)\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ l))\ (\mathsf{STATE}\ \sigma) = \quad \text{[until\_step\_thm]}$$
$$\quad \mathsf{if}\ \sigma.\mathsf{ip} = l\ \mathsf{then}$$
$$\quad\quad \mathsf{STATE}\ \sigma$$
$$\quad \mathsf{else}$$
$$\quad\quad ((\mathsf{step}\ p)\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ l))\ (\mathsf{step}\ p\ (\mathsf{STATE}\ \sigma)) \wedge$$
$$((\mathsf{step}\ p)\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ l))\ (\mathsf{FAULT}\ f) = \mathsf{FAULT}\ f$$

The proof of this theorem follows from a more general property of $\boldsymbol{\varepsilon}_1$.

$$\vdash \neg P\ 0 \wedge (\exists n \cdot P\ n) \Longrightarrow (\boldsymbol{\varepsilon}_1 n \cdot P\ n) = (\boldsymbol{\varepsilon}_1 n \cdot P\ (n+1)) + 1 \quad \text{[first\_suc\_thm]}$$

5 Equivalent Behavior

We now need to consider what it means for two programs to be equivalent. It may be too strong a requirement to insist that the behavior of an optimized program be identical to that of the original code. For example, consider the two programs presented in Fig. 2. If the address in register `r1` is not a valid destination, then both these programs will raise an ILLEGAL_STORE fault. Similarly, if the address in register `r4` is not a valid source, then both will raise an ILLEGAL_LOAD fault. If both these conditions hold then the unoptimized code will raise an ILLEGAL_STORE fault and the optimized code will raise an ILLEGAL_LOAD fault. Nevertheless, we might still consider these programs to be equivalent. More precisely, we will consider the behavior of two programs to be equivalent when they both raise faults, without insisting that they raise the same fault. Equivalence of behavior will therefore be defined on outcomes rather than simply being defined on states.

The programs shown in Fig. 1 are even more problematic. In the case where predicate register `p1` holds false then their behavior is the same, but when `p1` holds true then the optimized code writes data to register `r9` where the unoptimized code does not. This will be a problem unless `r9` is a scratch register, the contents of which are not of ongoing interest. Assuming that is the case, our notion of equivalence needs to be broadened to encompass programs with identical behavior across a nominated set of interesting registers.

We begin by defining an equivalence relation on register files that holds if some initial region of the register files are the same.

$$\vdash^{\mathsf{def}} (f_1 \cong_n f_2) = \quad \text{[grfile\_eq\_def]}$$
$$\quad n \leq \mathsf{sof} \wedge (\forall i\ x_1\ x_2 \cdot i < n \Longrightarrow \mathsf{reg\_read}\ x_1\ f_1\ i = \mathsf{reg\_read}\ x_2\ f_2\ i)$$

The following properties are important when reasoning about register files.[4]

$$\vdash f \cong_n f \quad \text{[grfile\_eq\_refl\_thm]}$$

[4] The equivalence relation $\cong_n$ is also symmetric and transitive as expected, but these properties are not used in the proofs described here.

$$\vdash (i \geq n) \Longrightarrow \qquad \text{[grfile\_eq\_above\_thm]}$$
$$((\mathsf{reg\_write}\ f_1\ i\ v) \cong_n f_2) = (f_1 \cong_n f2) \wedge$$
$$(f_2 \cong_n (\mathsf{reg\_write}\ f_2\ i\ v)) = (f_1 \cong_n f2)$$

Having defined an equivalence relation on register files, we can now define one on execution outcomes.

$$\vdash^{\mathsf{def}} (\mathsf{STATE}\ \sigma_1 \cong_n \mathsf{STATE}\ \sigma_2) = \qquad \text{[outcome\_eq\_def]}$$
$$(\sigma_1.\mathsf{mem} = \sigma_2.\mathsf{mem} \wedge \sigma_1.\mathsf{grfile} \cong_n \sigma_2.\mathsf{grfile} \wedge \sigma_1.\mathsf{prfile} = \sigma_2.\mathsf{prfile}) \wedge$$
$$(\mathsf{STATE}\ \sigma \cong_n \mathsf{FAULT}\ f) = \mathsf{F} \wedge$$
$$(\mathsf{STATE}\ \sigma \cong_n \bot) = \mathsf{F} \wedge$$
$$(\mathsf{FAULT}\ f \cong_n \mathsf{STATE}\ \sigma) = \mathsf{F} \wedge$$
$$(\mathsf{FAULT}\ f_1 \cong_n \mathsf{FAULT}\ f_2) = \mathsf{T} \wedge$$
$$(\mathsf{FAULT}\ f \cong_n \bot) = \mathsf{F} \wedge$$
$$(\bot \cong_n \mathsf{STATE}\ \sigma) = \mathsf{F} \wedge$$
$$(\bot \cong_n \mathsf{FAULT}\ f) = \mathsf{F} \wedge$$
$$(\bot \cong_n \bot) = \mathsf{T}$$

6 Example Proof

We can now return to a formal examination of the examples given in Sect. 2. We will consider only the example using data speculation, as its proof is the more challenging. We begin by specifying two instruction memories containing the original and optimized versions of the code from Fig. 2. Note that all instructions in these programs are unconditional, and hence predicated on register 0.

$$\vdash^{\mathsf{def}} \mathsf{original}\ 1000 = \mathsf{ST}\ 0\ 2\ 1\ 2 \wedge \qquad \text{[original\_def]}$$
$$\mathsf{original}\ 1001 = \mathsf{LD}\ 0\ 4\ 3\ 4$$

$$\vdash^{\mathsf{def}} \mathsf{optimized}\ 2000 = \mathsf{LD\_A}\ 0\ 4\ 3\ 4 \wedge \qquad \text{[optimized\_def]}$$
$$\mathsf{optimized}\ 2001 = \mathsf{ST}\ 0\ 2\ 1\ 2 \wedge$$
$$\mathsf{optimized}\ 2002 = \mathsf{CHK\_A}\ 0\ 3\ 4000 \wedge$$
$$\mathsf{optimized}\ 4000 = \mathsf{LD}\ 0\ 4\ 3\ 4 \wedge$$
$$\mathsf{optimized}\ 4001 = \mathsf{BR}\ 0\ 2003$$

The problem can now be stated as follows:

$$(\mathsf{STATE}\ \sigma_1 \cong_5 \mathsf{STATE}\ \sigma_2) \wedge$$
$$\mathsf{reg\_valid\_source}\ 1 \wedge \mathsf{reg\_valid\_source}\ 2 \wedge \mathsf{reg\_valid\_source}\ 4 \wedge$$
$$\sigma_1.\mathsf{ip} = 1000 \wedge \sigma_2.\mathsf{ip} = 2000 \Longrightarrow$$
$$(\mathsf{step\ original})\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ 1002)\ (\mathsf{STATE}\ \sigma_1) \cong_5$$
$$(\mathsf{step\ optimized})\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ 2003)\ (\mathsf{STATE}\ \sigma_2)$$

Note that equivalence between the executions can be proved only when the source registers are valid, as reading invalid registers returns unspecified results. Note also that the problem has been phrased using constant register names. We could also use variables to model symbolic register names, provided we add further assumptions asserting that the variables hold distinct values.

To start the proof, we substitute concrete records for the states σ_1 and σ_2. The assumptions allow us to select records with many common fields. We then separate out those assumptions that remain of interest, yielding the goal below:

$$
\begin{array}{l}
\bullet\ r_1 \cong_5 r_2 \\
\bullet\ \mathsf{reg\_valid\_source}\ 1 \bullet \mathsf{reg\_valid\_source}\ 2 \bullet \mathsf{reg\_valid\_source}\ 4 \\
\hline
\quad (\mathsf{step\ original})\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ 1002) \\
\qquad (\mathsf{STATE} \lhd \mathsf{ip}{:=}\,1000; \mathsf{mem}{:=}\,m; \mathsf{grfile}{:=}\,r_1; \\
\qquad\qquad \mathsf{prfile}{:=}\,p; \mathsf{alat}{:=}\,a_1; \mathsf{unknown}{:=}\,x_1 \rhd) \cong_5 \\
\quad (\mathsf{step\ optimized})\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ 2003) \\
\qquad (\mathsf{STATE} \lhd \mathsf{ip}{:=}\,2000; \mathsf{mem}{:=}\,m; \mathsf{grfile}{:=}\,r_2; \\
\qquad\qquad \mathsf{prfile}{:=}\,p; \mathsf{alat}{:=}\,a_2; \mathsf{unknown}{:=}\,x_2 \rhd)
\end{array}
$$

The records in this goal describe the symbolic state for both programs before any instructions have executed. Since neither program has reached its target instruction, we can use the theorems until_step_thm, step_def and inst_sem_def (see Sect. 4) to progress the symbolic execution of both programs as follows:

$$
\begin{array}{l}
\ldots \\
\hline
\quad (\mathsf{step\ original})\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ 1002) \\
\qquad ((\mathsf{st}\ 0\ 2\ 1\ 2)\ (\mathsf{STATE} \lhd \mathsf{ip}{:=}\,1001; \mathsf{mem}{:=}\,m; \mathsf{grfile}{:=}\,r_1; \\
\qquad\qquad\qquad \mathsf{prfile}{:=}\,p; \mathsf{alat}{:=}\,a_1; \mathsf{unknown}{:=}\,x_3 \rhd)) \cong_5 \\
\quad (\mathsf{step\ optimized})\ \mathsf{until}\ (\mathsf{at\_or\_fault}\ 2003) \\
\qquad ((\mathsf{ld\_a}\ 0\ 4\ 3\ 4)\ (\mathsf{STATE} \lhd \mathsf{ip}{:=}\,2001; \mathsf{mem}{:=}\,m; \mathsf{grfile}{:=}\,r_2; \\
\qquad\qquad\qquad\quad \mathsf{prfile}{:=}\,p; \mathsf{alat}{:=}\,a_2; \mathsf{unknown}{:=}\,x_4 \rhd))
\end{array}
$$

The values of the two unknown fields in the goal are actually expressions involving the next_unknown, the instruction, and the previous state. Since these fields contain no useful information, it is clearer if we generalize the proof by replacing them with fresh variables, as shown above.

The next step is to expand the definitions of st and ld_a. The definition of ld_a was presented in Fig. 3. These, and other, IA-64 instructions are defined as a selection among possible outcomes. We can use case analysis to reduce the resulting complex goal, that compares two conditionally defined outcomes, to a collection of simpler goals in which outcomes are compared under different premises. This step generates twenty subgoals, of which the following is among the most interesting.[5]

[5] The assumption $r_1 \cong_5 r_2$ has been used so that all reads refer to register file r_2.

```
• r1 ≅5 r2                                  • reg_valid_source 1
• reg_valid_source 2                        • reg_valid_source 4
• reg_valid_dest 3                          • ¬(reg_read x3 r2 1).nat
• ¬(reg_read x3 r2 2).nat                   • ¬(reg_read x4 r2 4).nat
• mem_valid_dest (reg_read x3 r2 1).val 2
• mem_valid_source (reg_read x4 r2 4).val 4
• ¬(mem_seq (reg_read x4 r2 4).val 4)
──────────────────────────────────────────────────────────────────────
 (step original) until (at_or_fault 1002)
   (STATE ◁ip = 1001;
          mem:=
            mem_write m (reg_read x3 r2 1).val 2 (reg_read x3 r2 2)
          grfile:= r1;
          prfile:= p;
          alat:= invalidate_multiple a1 (reg_read x3 r2 1).val 2;
          unknown:= x3▷) ≅5
 (step optimized) until (at_or_fault 2003)
   (STATE ◁ip:= 2001;
          mem:= m
          grfile:=
            reg_write r2 3 ◁val:= mem_read m (reg_read x4 r2 4).val 4;
                           nat:= F▷;
          prfile:= p;
          alat:= validate a2 3 (reg_read x4 r2 4).val 4;
          unknown:= x4▷)
```

Here the execution of both programs has progressed by one instruction, without encountering a fault. The goal has also accumulated a number of assumptions that will reduce the number of case splits needed for successive symbolic simulation steps. We repeat this process until each outcome in every goal is reduced to either FAULT or a STATE where the instruction pointer has reached the target. Each goal can then be reduced using outcome_eq_def (see Sect. 5). Because of the trivial equivalence of any two faulting outcomes, only four goals remain unsolved by this process.

Of the four subgoals that remain after symbolic simulation, two can be discharged by conditional rewriting with theorems about reading and writing registers and memory (see Sect. 3). This could be done as part of each symbolic simulation step, but it is faster if done just once at the end. The two remaining goals capture the heart of the problem, they hinge on the behavior of the ALAT.

In the first goal we see both programs have written data to register 3. The original program wrote the result of a read from memory, but the optimized program wrote the value zero. This must be the result of the advanced load having failed, causing a zero to be written, and the second load not having been performed. This should not happen, and indeed there is a contradiction in the assumptions. We have assumed that a check on register 3 in the ALAT succeeds, which is not possible since we have performed the operation invalidate_single on that register. This goal can be solved with the HOL Light model elimination procedure, MESON_TAC, using the ALAT definitions (see Sect. 3.4).

```
...
• check x'
    (invalidate_multiple (invalidate_single a2 3) (reg_read x3 r2 1).val 2) 3
------------------------------------------------------------------------------
  reg_write r1 3 ◁val:= mem_read ... (reg_read x3 r2 2) 4;nat:= F▷ ≅5
  reg_write r2 3 ◁val:= 0;nat:= F▷
```

In the second case, both programs have loaded register 3 with four bytes read from memory at the address held in register 2. However, the loads have been performed on different memories. In the unoptimized code, the memory was first modified by writing two bytes to the address held in register 1. The values loaded to register 3 will be the same provided the memory regions read and written do not overlap. This fact is embodied in an assumption of the goal.

```
...
• check x''
    (invalidate_multiple (validate a2 3 (reg_read x4 r2 4).val 4)
      (reg_read x3 r2 1 ).val 2) 3
------------------------------------------------------------------------------
  reg_write r2 3
    ◁val:= mem_read
              (mem_write m (reg_read x3 r2 1).val 2 (reg_read x3 r2 2).val)
              (reg_read x4 r2 4).val 4;
      nat:= F▷ ≅5
  reg_write r2 3 ◁val:= mem_read m (reg_read x4 r2 4).val 4;nat:= F▷
```

The assumption shown states that register 3 was set valid and associated with the memory region that was read. An invalidate_multiple operation was then performed, invalidating all registers with data read from regions overlapping the region of memory that was written. A check of register 3 then asserts that it is still valid, from which we can deduce that the regions of memory read and written do not overlap. We can prove this lemma using MESON_TAC on the definitions of the ALAT operations. Once proved, we can use it and the definition of mem_write (see Sect. 3.1) to solve the goal.

Both the examples presented in this paper were proved using HOL Light, as have other small examples using data speculation and transforming branching code into straight-line code using predication. All the proofs had the same form as the one just presented, in which the majority of the proof is completed by symbolic simulation and rewriting. None required any more user interaction to complete than was needed for the proof just presented.

7 Conclusion

This paper described a formal model for a significant portion of Intel's forthcoming IA-64 architecture. Theorems were proved about the model that allowed a symbolic simulator to be built using the HOL Light theorem prover. This system can be used to largely automate simple optimization proofs for assembly-level IA-64 code.

The scope of this research is intentionally limited. The problems considered are small, staying at the level of individual optimizing transformations rather than proofs

about entire programs. Likewise the properties proved are modest, checking only for equivalence between two similar programs rather than attempting to prove general correctness properties. By limiting the scope of the problem it was possible to find a solution that is largely automated. Indeed, the proofs could likely be more automated than they already are. The motivation for this approach comes from hardware verification where automated techniques with limited scope, like equivalence checking, have found industrial markets where more general interactive techniques have fared less well.

8 Future Work

One class of optimization not addressed by the work described here is software pipelining of loops. In these optimizations the original loop is transformed into a new loop where each cycle of the transformed loop executes instructions that correspond to steps within the execution of several successive iterations of the original loop. The transformation reduces data dependencies between instructions within the loop, thereby hiding the latency of the slower instructions. The term 'software pipelining' derives from an analogy with hardware pipelining, where each cycle executes steps from several successive instructions. The IA-64 includes several features that actively support the software pipelining of loops. We believe we can attack the problem of verifying transformations that pipeline a loop by building on the framework presented here using techniques analogous to those used to verify the equivalence of unpipelined and pipelined hardware implementations [1].

9 Related Work

In this paper we have demonstrated a system for verifying optimizing transformations to IA-64 assembly code. Perhaps the most closely related work is that of the Refinement Calculator project, which has built a general system to support program transformation and refinement in HOL [2, 10]. The work here differs from that in the following ways:

- Here we have worked with an unstructured assembly-level language, where as the Refinement Calculator (and similar transformation systems) manipulates structured programs.
- The work here has pursued a high degree of automation using symbolic simulation, where as systems like the Refinement Calculator usually focus on supporting a user-guided interactive style of reasoning.

Acknowledgements

This research was performed while the author was visiting the Microcomputer Software Laboratory of the Intel Corporation. Financial support was supplied by the Intel Corporation and the Australian Research Council. The author would like to thank John Harrison for his encouragement, support in visiting Intel, and his help with using HOL Light. Valuable feedback was provided by the anonymous referees.

References

1. Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Computer Aided Verification: Proceedings of the 6th International Conference (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, California, June 1994. Springer-Verlag.
2. Michael Butler, Jim Grundy, Thomas Långbacka, Rimvydas Rukšėnas, and Joakim von Wright. The refinement calculator: proof support for program refinement. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics and Theoretical Computer Science, pages 40–61, Victoria University of Wellington, New Zealand, March 1997. Springer-Verlag.
3. Carole Dulong. The IA-64 architecture at work. *Computer*, 31(7):24–32, July 1998.
4. Mike Gordon. A formalization of a simplified subset of the Alpha shared memory model. `http://www.cl.cam.ac.uk/Research/HVG/FTP/FTP.html#papers`.
5. John Harrison. *The HOL Light Manual*. University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, edition 1.0, May 1998.
6. Intel. IA-64 application developer's architecture guide. Order number 245188, Intel Corporation, Santa Clara CA, USA, May 1999.
7. Robert Brent Jones. *Applications of Symbolic Simulation to the Verification of Microprocessors*. PhD thesis, Stanford University, Department of Electrical Engineering, 161 Packard, 350 Serra Mall, Stanford CA 94305, USA, August 1999.
8. Michael Norrish and Konrad Slind. *The HOL System Description*. University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, hol98 Taupo-2 edition, February 2000.
9. SPARC International. *The SPARC Architecture Manual*. Prentice-Hall, New Jersey, 8th edition, 1992.
10. Joakim von Wright and Kaisa Sere. Program transformations and refinements in HOL. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *The HOL Theorem Proving and its Applications: International Workshop*, pages 231–239, University of California at Davis, August 1991. ACM-SIGDA, IEEE Computer Society Press.